An Architecture-Centric Approach to Detecting Security Patterns in Software

Michaela Bunke and Karsten Sohr

Technologie-Zentrum Informatik, Bremen, Germany,
{mbunke|sohr}@tzi.de

Abstract. Today, software security is an issue with increasing importance. Developers, software designers, end users, and enterprises have their own needs w.r.t. software security. Therefore, when designing software, security should be built in from the beginning, for example, by using security patterns. Utilizing security patterns already improves the security of software in early software development stages. In this paper, we show how to detect security patterns in code with the help of a reverse engineering tool-suite Bauhaus. Specifically, we describe an approach to detect the *Single Access Point* security pattern in two case studies using the hierarchical reflexion method implemented in Bauhaus.

1 Introduction

The increasing and diverse number of technologies that are connected to the Internet such as distributed enterprise systems or smartphones and other small electronic devices like the iPad brings the topic IT security to the foreground. We interact daily with these technologies and spend much trust on a well-established software development process. However, security vulnerabilities appear in the software on all kind of PC(-like) platforms. Hence, developers must more and more face security issues during software design and especially reengineering [26].

Today's security tools that aim to support developers during the implementation phase are based on static analysis. Chess et al. describe in depth how static analysis tools can detect bugs in code such as buffer overflows or simple race conditions [3]. These bugs can be used to exploit software. Using such analysis tools can increase the software's security by ensuring its stability through filtering out bugs at code level.

Beyond known static security analysis tools like Fortify [5], which work on source code, security modeling elements exist at the architectural level such as security patterns, which should enhance security at the software design stage. Security patterns can model security issues driven by the software's requirements. Similar to the design patterns introduced by Gamma et al. [6], security patterns describe a general reusable solution to a well-known problem. Yoder and Barcalow were the first that listed existing security patterns [29]. At that time, their popularity in the pattern community grew and many patterns have been published thereafter [12]. Yoshioka et al. [30] and Heymann et al. [12] give a good survey of the published security patterns till 2008. Design patterns and security patterns have several aspects in common, but are they similar?

VanHilst and Fernandez pointed out that “GoF Patterns are not security patterns” [27]. Like design patterns, which are also called *GoF patterns* [6], security patterns can be applied to implementing and structuring software systems, but they can also model security issues and security processes in enterprises such as “Enterprise Architecture Management Patterns” [4]. Based on this heterogeneity, the classification of security patterns is an important and often discussed issue [24, 28, 8, 9]. Different approaches exist, some describe basic topologies like separating the patterns in application domains (e.g., software, enterprise management), others describe complex layered structures. In this paper, we will have a closer look on patterns that describe how to implement a specific security feature, i.e., so-called structural patterns [8].

Within the pattern community, various descriptive models for security patterns exist [24]. The POSA model described by Buschmann et al. [2] is frequently used to describe the context and usage of security patterns. Some descriptions make use of UML diagrams or rarely source code snippets to clarify the security pattern modeling during the design phase. Nevertheless, security patterns are mostly described highly abstract; so it is difficult to understand the benefit or use if one is not familiar with software design linked with security issues [12]. Those circumstances have an impact on the software design when you have to ensure security interests and select an appropriate security pattern for the software needs. Halkidis et al. show that the usage of security patterns can offer a reasonable protection against most common attacks [10].

Frequently, software has to be modified due to changing requirements, bugs and security flaws. Moreover, the reconstruction of patterns in grown systems is quite difficult. Maintenance programmers, however, must deal with such use cases. Given that patterns in general are the best-known solution of a recurring problem [6], security patterns should also be recognized during the maintenance process to guarantee security objectives and requirements. For example, a detected *Check Point* pattern [24] allows one to conclude that on this point in code relevant information will be validated before further steps in the application flow are carried out. If the developer knows about a security pattern in code, he is aware of what is going on in this code unit or component and can program according to this context. An improvement of software quality caused by detecting incorrect or not accomplished security patterns is conceivable. In this case, the developer can avoid bypassing this *Check Point* in further implementations.

For this maintenance process, there exist several reengineering tools to get a clear view about the software structure and behavior. However, only a few of those tools take the well-known design patterns into account to support program comprehension at that point. Some approaches are presented in [27]. Presently none of them support the detection of security patterns.

All shown factors do not support the application, comprehension and recognition of security patterns in software. Therefore, it is desirable to integrate security patterns with a program comprehension tool. This ensures that security patterns are preserved during software maintenance process and their highlighting allow once well-directed implementations of new (security) features in software.

In this paper, we focus on the reengineering scope by discussing security patterns. Specifically, we use the Resource Flow Graph (RFG) representation provided by the reverse engineering tool-suite Bauhaus [21]. With this program representation, we are able to use the integrated program comprehension method called hierarchical reflexion method [16]. This method aims to reconstruct a software architecture by mapping a hypothetical architecture to the actual software architecture extracted from the source code. Our goal is to depict the existence of security patterns on an abstract architectural level. We also describe which methods can help programmers in preserving security patterns during the software maintenance process. To demonstrate the feasibility of our approach, we present two case studies. Here, security patterns are identified in a software architecture by using our program comprehension technique.

The remainder of this paper is structured as follows. In Section 2, we briefly describe the software analysis tool Bauhaus. There we will concentrate on the RFG and the hierarchical reflexion method. In Section 3, we present further steps in combining an architecture-based methodology with security pattern detection. This is followed by the description of the case studies in Section 4. After discussing related work, we give an outlook in Section 6.

2 The Bauhaus Tool

The Bauhaus tool-suite is a reverse engineering tool-suite that has been employed in several industry projects [21]. Bauhaus allows one to retain two abstractions from the source code. The low-level representation called Intermediate Language (IML) is an attributed syntax tree (an enhanced AST) that contains the detailed program structure information such as loop statements, variable definitions and name bindings. The RFG, a more abstract representation, works at a higher abstraction level and represents architecturally relevant information of the software. At present such a graph can be created for programs written in C, C++, Java, ADA and C#. The RFG is a hierarchical graph that consists of typed nodes and edges representing elements like types, components, and routines and their relations. The RFG's information is stored in structured in *views*, where each view represents a different aspect of the architecture, e.g. a call graph.

Several analyses are built upon this infrastructure to derive design and architectural information like the so-called *hierarchical reflexion analysis* [16]. This analysis extends the original analysis developed by Murphy et al. [18] to hierarchical systems. It starts with a hypothesis of the architecture and a mapping of existing implementation components onto architectural components provided by an human analyst. An automated analysis then determines convergences and differences among the architecture and the implementation model, the so-called *reflexion model*. Based on these findings, the architecture and mapping may be refined and the process will be repeated until the architecture model sufficiently describes the implementation.

Usually, this procedure will be used when a software system has to be modified but the documentation or the knowledge of it got lost. Besides this reconstruction the reflexion analysis can be used to check the present implementation against their architectural specification.

3 Security Aspects and the RFG

Sohr and Berger [25] depict some possibilities to accomplish a security analysis with the RFG. We resume on their point and discuss other security aspects that can be based upon the RFG. In contrast to their ideas, we will not focus on policies and RBAC extensions. Our focus is software quality assurance and program comprehension in conjunction with security patterns.

Hierarchical reflexion method for security issues: As already mentioned in Section 2, the hierarchical reflexion analysis is a well-known method to reconstruct software architectures. This method can be used to identify security patterns. These will be marked as potential patterns and can be used to show deficiencies in the software architecture. With an automated check against the real source code, there can be detected architecture violations such as calling a component not through a *Check Point* that can induce to security concept violations. This reflexion method is used in case studies, presented in Section 4 to detect a selected security pattern in a software's architecture.

Security patterns at the architecture level: As shown above, the RFG provides the ability to create new views. These views can be created containing only elements focusing on special purposes. Conceivable is a view containing elements that are supposed to belong to a single security pattern, possibly identified by the method described above. If one has identified more than one security pattern and created them on different views, one can create a new view by intersecting or uniting the view to visualize composed or merged security patterns as described in [23]. Possibly this process can be automated when the system knows several available or often occurring compositions of security patterns. Presenting such combinations to maintenance programmers may facilitate the realization of adequately and inadequately programmed pattern collaborations. For instance, consider the combination of *Single Access Point* and *Check Point* pattern [24], where maybe a badly implemented cooperation raise security leaks.

Automatic detection, suggestions and learning: Semi-automatic detection of security patterns is good, but is time-consuming and requires deep knowledge of the system. A better approach would be the automatic detection. However, this is not easy to realize as there are many challenges as described in Section 1. Maybe, there exists a pattern language that fulfills our needs for the description of security patterns at the architecture level. We then can transform these descriptions to the RFG model for an automatic pattern matching and are able to present the maintenance programmer pattern suggestions. These suggestions can be assessed or modified by the programmer to improve the security pattern model. Thereby, we can collect pattern derivatives for improving the automatic or even the semi-automatic detection. Moreover, this collection gives the abstract appearance of security patterns a more clear shape that can be reused. This technique can be refined by using security anti- or misuse patterns to model an architecture's irregularities to be able to detect the incorrect usage of security patterns. The benefit of the sketched technique is that software systems can be post-checked and hardened before they will be released.

Source and sink markers for pattern endings: Information flow is an important issue concerning software security. A security view of the RFG can also model fractals of the information flow. If we combine the RFG with the other more-detailed code representations such as the IML, we can model the information flow between components. If we have detected a security pattern or compound, we can extract the pattern in a new view and highlight sources and sinks of the patterns. This would enhance the role-based view described by Sohr et al. [25] and give the opportunity to plug in a further information flow analysis to validate the pattern's behavior. To give an example consider the communication with a database or a password manager. In these cases, the visualization of security patterns' sinks and sources like architectural glue dots addresses the maintenance programmer. It will support and ensure the information flow comprehension while reconstructing the software system.

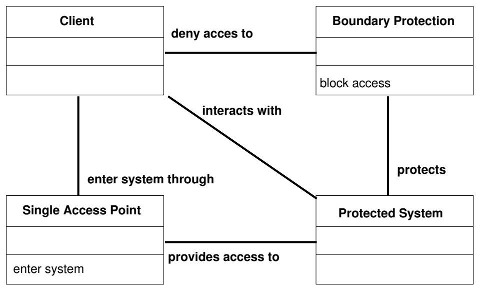
4 Early Case Studies

We now discuss our architecture-centric security pattern analysis in the context of two case studies. We selected the security pattern *Single Access Point* to demonstrate that we can identify this pattern within an abstract software representation.

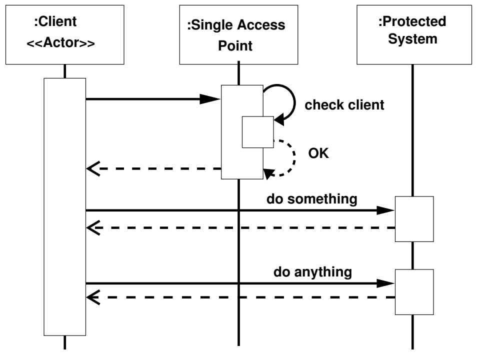
We chose primarily the open source instant messenger client Spark [13] and an open source Android application named Simple Android Instant Messaging Application [17]. Both are Java-based programs, so we took the Java byte code and generated the software architecture in the RFG format. This is the starting point for using the hierarchical reflexion method to detect the *Single Access Point* pattern. First of all, we present the pattern in Section 4.1 and then we will have a closer look on the case studies in 4.2 and 4.3.

4.1 Single Access Point Pattern

A *Single Access Point* pattern [24] provides access to a system for external clients. Moreover, it ensures that the system cannot be damaged or misused by such clients. The idea behind this pattern is that an exclusive door to the system can be better protected and controlled than many. Fig. 1 depicts the UML diagrams for the *Single Access Point* security pattern. For this reason, many application clients such as twitter or instant messenger clients that provide any kind of access to other systems use a derivative of this pattern in order to provide clients (mostly, users) access to underlying services.



(a) *Single Access Point* UML diagram

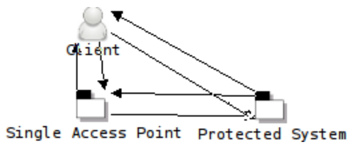


(b) *Single Access Point* Sequence Diagram

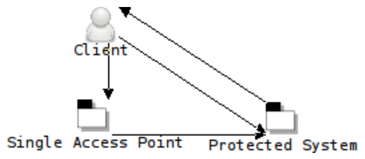
Fig. 1. *Single Access Point* [24]

According to the UML diagram in Fig. 1(a), we modeled a first hypothetical architecture (see Fig. 2(a)) containing the components `Client`, `Single Access Point`, and `Protected System`. The client that uses this software is represented by the `Client` component in the architecture. Given that the reflexion method is based on static code dependencies and our client in the case studies is a human, we obviously will never see any match in the dependencies with the system. Schumacher et al. mentioned that the `Boundary Protection` component of a protected system was often hard to show. This also applies to our case studies where the client is a human user that needs to know a user name and password to gain access to the protected system. In this case, the user's knowledge models the `Boundary Protection` component. Hence we skip this component as we cannot model it according to static analysis. We have not specified in depth the dependencies between components such as "calls", "references", and "inherits" to simplify the dependencies and modeled them as undirected dependencies according to the undirected edges in Fig. 1(a).

After our first attempt to model the architecture, we realized that the UML model was not adequate enough to represent the idea of the *Single Access Point* pattern. Thus, not every direction of the dependencies between the components may be allowed to ensure a secure behavior. For this reason, we give a more specific access point model according to the information given in Fig. 1(b). First, the `Client` interacts with the `Single Access Point` component, and thereafter the client can interact with the `Protected System`. Based upon this, we assume that the `Single Access Point` component allows or denies the user's request and informs the `Protected System` about the response. Possibly, the `Single Access Point` component instantiates a further window to allow the user to



(a) Hypothetical architecture



(b) Hypothetical architecture - specific dependencies

Fig. 2. Hypothetical architectures

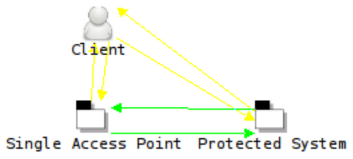
interact with the `Protected System` after the logon. Therefore, we specify that a proper behavior of the system is that the `Single Access Point` has dependencies to the `Protected System` to call, communicate or instantiate something after passing the `Single Access Point`. The corresponding hypothetical architecture is shown in Fig. 2(b).

Both hypothetical architectures will be used with the hierarchical reflexion method on the chosen applications to show and discuss distinctive features.

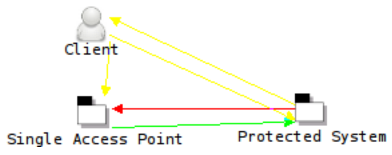
4.2 Case Study: Spark

Spark is an open source instant messenger client that provides a login screen and is expected to use this pattern [13]. It is a client that allows users to log on to an instant messenger network and then receive and write instant messages to other users.

Intuitively, we map the software components according to their names such as "LoginDialog", "LoginSettingsDialog" to the component `Single Access Point`. Then, we assume that the rest of the code is in package "org.jivesoftware" is the `Protected System`.



(a) Model check result against Fig. 2(a)



(b) Model check result against Fig. 2(b)

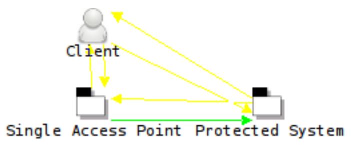
Fig. 3. Detection results - Spark

Fig. 3(a) shows the match between the hypothetical architecture and the real code architecture. As expected, outgoing and incoming edges of the `Client` component are marked as absence (yellow edges). The edges between `Single Access Point` and `Protected System` are marked as convergences. This shows that this pattern can be found in the software's architecture by using the reflexion method.

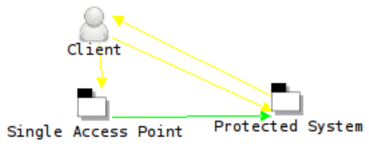
In Fig. 3(b), we try to detect a login behavior in Spark. The architecture match result is depicted in Fig. 3(b). The expected dependency between `Single Access Point` and `Protected System` is marked as convergence (green edge). However, there exist more dependencies than we have modeled, represented by the red edges. They arise from static field usages and class instantiations. This shows that the two identified components are bundled together and are not strictly separated in Spark as one might expect according to their task.

4.3 Case Study: Simple Android Instant Messaging Application

According to our experiences with Spark we will have a closer look on another open source application. The Simple Android Instant Messaging Application [17] is an example application for the mobile phone platform Android [7]. The author's intention to make this application freely available was to provide interested people with an example of an Android application and show how instant messaging can be provided easily. It communicates via the http protocol with a web server. This server is also used for user authentication. We assume that every component starting or ending with “Login” will indicate the `Single Access Point` in the source code. The rest belongs to the `Protected System` because it provides the instant messaging communication with the server. For the reflexion method, we used the same hypothetical architectures as depicted in Fig. 2(a) and 2(b).



(a) Model check result checking against the hypothetical architecture from Fig. 2(a)



(b) Model check result checking against the hypothetical architecture from Fig. 2(b)

Fig. 4. Detection results - Simple Android Instant Messaging Application

Fig. 4(b) depicts the architecture match result for the expected behavior. Here, the dependency between the components `Single Access Point` and `Protected System` is marked as convergence (green edge) and the dependencies to the client are marked as absence (yellow edges). This indicates that in the Simple Android Instant Messaging Application the components for `Single Access Point` and `Protected System` are separated in the code. This hypothesis is confirmed by Fig. 4(a) that shows an absence between `Protected System` and `Single Access Point`.

4.4 Conclusion

We demonstrated with these case studies that we were able to detect a security pattern within software using the components described in the pattern description. Besides the two case studies discussed, we have used this method to detect the *Single Access Point* pattern in two other software systems and to detect the *Runtime mix'n and match* design pattern [1] that is coupled with a *Check Point* security pattern. In particular, we detected it in the middleware of the open source platform Android [7].

Towards this case study we expect to be able to analyse more security patterns which apply the classification of structural patterns. They description should also contain UML diagrams that clarify their structure and behavior. However, shown in this study even with the help of such diagrams it cannot be clearly decided whether a security pattern is modeled accurate or inaccurate. In our case Spark models the *Single Access Point* pattern according to the UML class diagram and the Simple Android Instant Messaging Application in compliance with both UML diagrams. Hence further researches on security patterns and their appearance in software architecture are reasonable.

With the introduced static examination, however, we are not able to consider if the system behaves in the expected way. Therefore, we will need more source code information as provided in the IML representation, a more specific static description of this pattern or even dynamic analysis information. Moreover, on detecting such patterns automatically or semi-automatically we have to deal with abstract descriptions that must be modeled differently for several application contexts. For example, in the shown case the client was a user that must enter his credentials. In another case, the client is possibly a web service that tries to use another web service.

5 Related Work

There exist a plethora of works for the static security analysis of software [3]. The works on static analysis for security often use the source code in order to detect common vulnerabilities such as buffer overflows or cross site scripting [5, 20]. Another approach combines type-based security and annotations with dependence graph-based information flow control [11]. All aforementioned approaches do not deal with security patterns.

In addition, VanHilst and Fernandez [27] discuss the possibilities to detect security patterns using reverse engineering like [19]. They identify some problems that may occur during detection, but they do not describe a practical approach using the reflexion method. Concerning security and software architectures, Ryoo et al. [22] presented a basic approach to detecting architectural constructs and properties that make software less secure. Another idea on employing the software architecture for static security analysis was described by Sohr and Berger [25]. They use the RFG to check policies and permissions on Java EE and the Android platform. An approach to the automated verification of UMLsec models has been presented by Jürjens and Shabalin [14]. They present automated verification of UML models w.r.t. modeled security requirements. The precondition to this is that there exists a UML-modeled architecture for the software system. Our approach has the reverse engineering point of view, and we work with an abstract representation based on the software implementation for searching security patterns. Thus, our approach requires no UML documentation of the software architecture and represents always the actual software's implementation state.

6 Outlook

In this paper, we demonstrated that we are able to detect security patterns using the reflexion method. As indicated in Section 4.4, we plan to improve on the automation degree of our detection process for instance using the incremental reflexion method [15]. In addition, we will search for the best arrangement of static and dynamic analysis techniques to support this goal. A further step in our work will be the examination of larger software systems to point out which security patterns are used and clarify their impact on software's architecture.

Section 1 indicates that many topics are open to discuss security patterns and reverse engineering. Thus we consider this work as a starting point for further approaches in security and program comprehension, bringing together the different research communities of reverse engineering and software security.

References

1. Paul G. Austrem. Runtime mix'n and match design pattern. In *Proc. of the 15th Pattern Languages of Programs*, pages 1–8, New York, NY, USA, 2008. ACM.
2. Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture: A System of Patterns.* Wiley, Chichester, UK, 1996.
3. Brian Chess and Gary McGraw. Static analysis for security. *IEEE Security and Privacy*, 2:76–79, 2004.
4. Alexander M. Ernst. Enterprise architecture management patterns. In *Proc. of the 15th Pattern Languages of Programs*, pages 1–20, New York, NY, USA, 2008. ACM.
5. Fortify Software. Fortify source code analyser, 2009. `http://www.fortify.com/products`.
6. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Object-Oriented Software.* Addison Wesley, 1995.
7. Google Inc. Android development, 2010. `http://developer.android.com/index.html`.
8. Munawar Hafiz, Paul Adamczyk, and Ralph E. Johnson. Organizing security patterns. *IEEE Software*, 24:52–60, 2007.
9. Munawar Hafiz and Ralph Johnson. Security patterns and their classification schemes. Technical report, Technical Report for Microsoft's Patterns and Practices Group, September 2006.

10. Spyros T. Halkidis, Alexander Chatzigeorgiou, and George Stephanides. A qualitative analysis of software security patterns. *Computers & Security*, 25(5):379–392, 2006.
11. Christian Hammer. Experiences with pdg-based ifc. In F. Massacci, D. Wallach, and N. Zannone, editors, *Proc. of International Symposium on Engineering Secure Software and Systems*, volume 5965 of *LNCS*, pages 44–60. Springer-Verlag, February 2010.
12. Thomas Heyman, Koen Yskout, Riccardo Scandariato, and Wouter Joosen. An analysis of the security patterns landscape. In *Proc. of 3rd International Workshop on Software Engineering for Secure Systems*. IEEE Computer Society, 2007.
13. Jive Software. Spark - project page. Online, 2010. `http://www.igniterealtime.org/projects/spark/index.jsp`.
14. Jan Jürjens and Pasha Shabalin. Automated verification of UMLsec models for security requirements. In *Proc. of UML 2004 - The Unified Modelling Language: Modelling Languages and Applications*, volume 3273 of *LNCS*, pages 365–379. Springer, 2004.
15. Rainer Koschke. Incremental reflexion analysis. In *European Conference on Software Maintenance and Reengineering*. IEEE Computer Society Press, 2010.
16. Rainer Koschke and Daniel Simon. Hierarchical reflexion models. In *Proc. of 10th Working Conference on Reverse Engineering*, pages 36–45, nov. 2003.
17. Ahmet Oguz Mermerkaya. Simple android instant messaging application - project page, 2010. `http://code.google.com/p/simple-android-instant-messaging-application/`.
18. Gail C. Murphy, David Notkin, and Kevin J. Sullivan. Software reflexion models: Bridging the gap between design and implementation. *IEEE Transactions on Software Engineering*, 27(4):364–380, April 2001.
19. Jörg Niere, Wilhelm Schäfer, Jörg P. Wadsack, Lothar Wendehals, and Jim Welsh. Towards pattern-based design recovery. In *Proc. of the 24th International Conference on Software Engineering*, pages 338–348. ACM, 2002.
20. Ounce Labs Inc. Website, 2010. `http://www.ouncelabs.com/`.
21. Aoun Raza, Guther Vogel, and Erhard Plödereder. Bauhaus—A tool suite for program analysis and reverse engineering. In *Ada-Europe*, LNCS, pages 71–82. Springer, 2006.
22. Jungwoo Ryoo, Phil Laplante, and Rick Kazman. In search of architectural patterns for software security. *Computer*, 42:98–100, 2009.
23. Markus Schumacher. Merging security patterns. In *Proc. of 6th European Conference on Pattern Languages of Programs*, 2001. `http://www.voelter.de/data/workshops/europlop2001/merging_security_patterns.pdf`.
24. Markus Schumacher, Eduardo Fernandez, Duane Hybertson, and Frank Buschmann. *Security Patterns: Integrating Security and Systems Engineering*. John Wiley & Sons, 2005.
25. Karsten Sohr and Bernhard Berger. Towards architecture-centric security analysis of software. In *Proc. of International Symposium on Engineering Secure Software and Systems*. Springer-Verlag, 2010.
26. The H Security. Number of critical, but unpatched, vulnerabilities is rising. Online, 2010. `http://www.h-online.com/security/news/item/Number-of-critical-but-unpatched-vulnerabilities-is-rising-1067495.html`.
27. Michael VanHilst and Eduardo B. Fernandez. Reverse engineering to detect security patterns in code. In *Proc. of 1st International Workshop on Software Patterns and Quality*. Information Processing Society of Japan, December 2007.

28. Hironori Washizaki, Eduardo B. Fernandez, Katsuhisa Maruyama, Atsuto Kubo, and Nobukazu Yoshioka. Improving the classification of security patterns. *Workshop on International Conference on Database and Expert Systems Applications*, 0:165–170, 2009.
29. Joseph Yoder and Jeffrey Barcalow. Architectural patterns for enabling application security. In *Proc. of 4th Pattern Languages of Programs*, Monticello/IL, 1997.
30. Nobukazu Yoshioka, Hironori Washizaki, and Katsuhisa Maruyma. A survey on security patterns. *Progress in Informatics*, 5:35–47, 2008.